Assignment #4 **Authors:** Ryan Ceresani Class: 605.744 Information Retrieval **Term:** Fall 2021 This assignment submission is structured slightly different than my previous - as we are using a Notebook in place of a "main" application. The notebook provides a nice integration for intermediate code output, plotting, results and experimentation. **Expected Deliverables**  Report outlining methodologies, tools used, parameter decisions, etc. • Precision, Recall, F1 score for the "dev" dataset using "Title" column as features. Metrics for "dev" dataset using "Title", "Abstract", and "Keywords" as features. • Perform an additional non-trival experimentation. Predictions for the "test" dataset. Results First Reporting We will dive into all of the details later but will jump straight into the results. Specific details for each run will be visible in the Notebook cells below. Initial Attempt: SingleLinearLayerModel Data Set Details Initialized Embedding that was Trained only on the Training Dataset Used only "Title" as feature. Training Details ■ Epochs = 50 ■ Learning Rate = 0.001 Results Precision: 56 / (56 + 348) = **0.1386** • Recall: 56 / (56 + 94) = 0.3733■ F1: 2 (0.1386 0.3733) / (0.1386 + 0.3733) = **0.2021** Avg Precision: 0.07113 Experimentation: GloVeMLP Because of bad results - did experimentation before moving on to using "Title" "Abstract" "Keywords" Data Set Details Used only "Title", "Abstract", "Keywords" as features. GloVe 840B.300d used as vocab + initalized embeddings for model. Training Details Same as above Results Precision: 126 / (126 + 580) = 0.1785 ■ Recall: 126 / (126 + 24) = **0.84** ■ F1: 2 (0.1785 0.84) / (0.1785 + 0.84) = **0.2944** AP: 0.15486346777255336 Overall Methodology Before getting into code specifics, we will address some overall elements used during this assignment. The main approach I am choosing to perform Text Classification is through deep learning. **Open Source Libraries** To facility deep learning, this experiment makes heavy use of PyTorch and torchtext for doing the underlying operations. Additionally, scikit-learn is useful for their metrics library which offers useful calculation ability for a variety of classifation metrics (whether you use one of their estimators or not.) PyTorch: This is the heavy-lifter for a number of backing abstract classes performing a variety of functions. Torchtext: Extension off the official PyTorch to provide text based utilities. vocab : This generates a Vocabulary object from an iterator which can be used to map words to indices, converting tokens utils: It also has some built in utilities for tokenizing and creating ngrams. • scikit-learn: Used for the metrics library to generate scores and repots. **Dataset** Systematic Review • NOTE: The Document hash:a8113f0b-6561-3178-8c2d-7b4ebac229ff contained an odd sequence of characters in UTF8 at the beginning of the "Article" section which caused problems for the python stdlib csv\_reader. The value was sanitized to remove the characters (", ) - which would be removed in tokenization anyway. **Dataset Challenges** A primary challenge imposed by this dataset is the vast class imbalance within training. (30:1 negative skew) To counteract this, two approaches considered were: 1. The DataLoader used a WeightedRandomSampler which was weighted to allow the random sampler topopulate batches with a statistically equal chance for both classes. 2. The chosen loss function (CrossEntropyLoss) used inverse class frequency weighting to encourage learning on the minority class. This ended up providing less benefit and the combination of both was worse. Models After numerous experimentation, two models were used during the course of this. Both used an EmbeddingBag instead of a typical embedding. It provides a lot of functional benefits but does remove the ability to consider sequences of words. So the model loses all context of the sentences themselves. The actual code for the models will be appended to the end of the notebook. Simple Linear Model The first draft model used the embedding bag and a single Linear Layer. The results were not very promising and it was abandoned. (It was tested on the TAK features as well with limited change.) **MLP Model** In an effort to increase the generalization, which was a problem with the first model, I created a straight-forward MLP. The inclusion of activations, in this case, GELU, provided for non-linearity and seemed to increase performance. • 3 fully-connected layers GELU Activation • Dropout during training **Custom Modules - Source Code** A number of custom code was generated to support this experiment. datasets.py import io from typing import Callable, List import torch from torch.utils import data from torch.utils.data.sampler import WeightedRandomSampler from torchtext.data.utils import get\_tokenizer from torchtext.utils import unicode\_csv\_reader from torchtext.vocab import Vocab \_default\_tokenizer = get\_tokenizer("basic\_english") DEFAULT\_LABEL\_TRANSFORM = lambda x: x DEFAULT\_TEXT\_TRANSFORM = lambda x: \_default\_tokenizer(x) def create\_torch\_dataloader( dataset: data.Dataset, vocab: Vocab, label\_transform: Callable = DEFAULT\_LABEL\_TRANSFORM, text\_transform: Callable = DEFAULT\_TEXT\_TRANSFORM, weighted=True, \*\*kwargs ): """Creates a Pytorch style dataloader using a dataset and a precompiled vocab. The dataset returns "model-ready" data. Args: dataset: The raw text dataset to be used during inference vocab: the premade vocabulary used to index words/phrases label\_transform: any operation used on the datasets label output text\_transform: operation used on the raw text sentence outputs from the data weighted: whether to weight the samples based on class distribution \*\*kwargs: any additional kwargs used by Pytorch DataLoaders. Returns: A PyTorch DataLoader to be used during training, eval, or test. device = torch.device("cuda" if torch.cuda.is\_available() else "cpu") def \_collate\_batch(batch): label\_list, docid\_list, text\_list, offsets = [], [], [], [0] for (\_label, \_docid, \_text) in batch: label\_list.append(label\_transform(\_label)) processed\_text = torch.tensor( vocab(text\_transform(\_text)), dtype=torch.int64 text\_list.append(processed\_text) offsets.append(processed\_text.size(0)) docid\_list.append(\_docid) label\_list = torch.tensor(label\_list, dtype=torch.int64) offsets = torch.tensor(offsets[:-1]).cumsum(dim=0) text\_list = torch.cat(text\_list) return label\_list.to(device), text\_list.to(device), offsets.to(device), docid\_list if weighted: weights = dataset.sample\_weights sampler = WeightedRandomSampler(weights=weights, num\_samples=len(weights)) else: sampler = None return data.DataLoader( dataset, collate\_fn=\_collate\_batch, shuffle=(sampler is None), sampler=sampler, \*\*kwargs ) class TSVRawTextIterableDataset(data.IterableDataset): """Dataset that loads TSV data incrementally as an iterable and returns raw text. This dataset must be traversed in order as it only reads from the TSV file as it is called. Useful if the size of data is too large to load into memory at once. def \_\_init\_\_(self, filepath: str, data\_columns: List[int]): """Loads an iterator from a file. Args: filepath: location of the .tsv file data\_columns: the columns in the .tsv that are used as feature data self.\_number\_of\_items = \_get\_tsv\_file\_length(filepath) self.\_iterator = \_create\_data\_from\_tsv( filepath, data\_column\_indices=data\_columns ) self.\_current\_position = 0 def \_\_iter\_\_(self): return self def \_\_next\_\_(self): item = next(self.\_iterator) self.\_current\_position += 1 return item def \_\_len\_\_(self): return self.\_number\_of\_items class TSVRawTextMapDataset(data.Dataset): """Dataset that loads all TSV data into memory and returns raw text. This dataset provides a map interface, allowing access to any entry. Useful for modifying the sampling or order during training. def \_\_init\_\_(self, filepath: str, data\_columns: List[int]): """Loads .tsv structed data into memory. Args: filepath: location of the .tsv file data\_columns: the columns in the .tsv that are used as feature data self.\_records = list( \_create\_data\_from\_tsv(filepath, data\_column\_indices=data\_columns) self.\_sample\_weights, self.\_class\_weights = self.\_calculate\_weights() @property def sample\_weights(self): return self.\_sample\_weights @property def class\_weights(self): return self.\_class\_weights def \_calculate\_weights(self): targets = torch.tensor( [label if label > 0 else 0 for label, \*\_ in self.\_records] unique, sample\_counts = torch.unique(targets, return\_counts=True) weight = 1.0 / sample\_counts sample\_weights = torch.tensor([weight[t] for t in targets]) class\_weights = weight / weight.sum() return sample\_weights, class\_weights def \_\_getitem\_\_(self, index): return self.\_records[index] def \_\_len\_\_(self): return len(self.\_records) def \_create\_data\_from\_tsv(data\_path, data\_column\_indices): with io.open(data\_path, encoding="utf8") as f: reader = unicode\_csv\_reader(f, delimiter="\t") for row in reader: data = [row[i] for i in data\_column\_indices] yield int(row[0]), row[1], " ".join(data) def \_get\_tsv\_file\_length(data\_path): with io.open(data\_path, encoding="utf8") as f: row\_count = sum(1 for row in f) return row\_count **Experiment Walkthrough** We will now walk through the experiment notebook to see results in action. **Imports** The open source and custom modules used are imported first. In [2]: %load\_ext autoreload %autoreload 2 In [1]: import torch from torch.utils.tensorboard import SummaryWriter from torchtext.data.utils import ngrams iterator from torchtext.data.utils import get tokenizer In [2]: from ir classification import datasets, models from ir classification import vocab as ir vocab from ir classification import train **Setup the Training and Validation Datasets** This first time through we do not use GloVe, just the original vocab. In [3]: datafield map = {"assessment": 0, "doc id": 1, "title": 2, "authors": 3, "journal": 4, "issn": 5, "year": 6, "] data columns = [datafield map["title"]] ngrams = 1batch size = 64 # Create vocab from the training data. vocab = ir vocab.create vocab from tsv("../datasets/systematic review/phase1.train.shuf.tsv", data columns, ngr # Load the TSV into datasets with the appropriate feature columns. train\_dataset = datasets.TSVRawTextMapDataset("../datasets/systematic\_review/phase1.train.shuf.tsv", data\_colum val dataset = datasets.TSVRawTextMapDataset("../datasets/systematic review/phase1.dev.shuf.tsv", data columns) # Create the transforms for the dataloader to appropriately format the contents of the files. label transform = lambda x: x if x > 0 else 0 tokenizer = get tokenizer("spacy") text transform = lambda x: list(ngrams iterator(tokenizer(x), ngrams)) # Instantiate the dataloaders. train dataloader = datasets.create torch dataloader(train dataset, vocab, label transform, text transform, wej val dataloader = datasets.create torch dataloader(val dataset, vocab, label transform, text transform, weighted C:\Users\ryanc\AppData\Local\pypoetry\Cache\virtualenvs\ir-classification- Pgcz6ju-py3.9\lib\site-packages\torc htext\data\utils.py:123: UserWarning: Spacy model "en" could not be loaded, trying "en core web sm" instead warnings.warn(f'Spacy model "{language}" could not be loaded, trying "{OLD MODEL SHORTCUTS[language]}" instea Instantiate the Model In [4]: num classes = 2vocab size = len(vocab) # from vocab created earlier. embedding size = 64 hidden layer size = 100 # Enable compatability when training with GPU enabled devices. # (Some development work was done in Google Colab with GPU) device = torch.device("cuda" if torch.cuda.is available() else "cpu") model = models.EmbeddingBagLinearModel(vocab size, embedding size, num classes).to(device) Setup the top-level training loop The custom code was meant to handle the individual step and epoch levels generically. This setup should let us change the components experimentally in cells like below without much other hassle. In [5]: EPOCHS = 50learning rate = 0.001 # Create the loss function weighted to inverse class distribution # loss function = torch.nn.CrossEntropyLoss(weight=train dataset.class weights) loss function = torch.nn.CrossEntropyLoss() # Instantiate a Stochastic Gradient Descent optimizer and "Auto" Learning Rate schedule. optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate) scheduler = torch.optim.lr scheduler.StepLR(optimizer, 1.0, gamma=0.95) # Tensorboard writing utility class. log dir = "runs/SimpleLinearModel" writer = SummaryWriter(log dir=log dir) # Perform Training for i in range(EPOCHS): start iter = len(train dataloader) \* i train.train epoch(i, model, optimizer, loss function, train dataloader, start iter=start iter, writer=write validation\_results = train.evaluate\_epoch(i, model, loss\_function, val\_dataloader, writer) scheduler.step() torch.save(model.state dict(), "model weights/title only SLM state dict.pth") Epoch 0: 100% | 340/340 [00:08<00:00, 40.23 batch/s, accurracy=1, loss=0.459] Validation: 0: 100% | 76/76 [00:00<00:00, 89.42 batch/s, accurracy=0.82, loss=0.413] Epoch 1: 100%| | 340/340 [00:07<00:00, 48.20 batch/s, accurracy=1, loss=0.268] 76/76 [00:00<00:00, 140.39 batch/s, accurracy=0.92, loss=0.266] Validation: 1: 100%| Epoch 2: 100%| | 340/340 [00:06<00:00, 49.98 batch/s, accurracy=1, loss=0.126] Validation: 2: 100%| | 76/76 [00:00<00:00, 127.11 batch/s, accurracy=0.74, loss=0.514] Epoch 3: 100%  $\parallel$  340/340 [00:07<00:00, 47.85 batch/s, accurracy=1, loss=0.0499] Validation: 3: 100%| 76/76 [00:00<00:00, 140.08 batch/s, accurracy=0.86, loss=0.308] Epoch 4: 100%| 340/340 [00:06<00:00, 50.22 batch/s, accurracy=1, loss=0.0969] 76/76 [00:00<00:00, 132.20 batch/s, accurracy=0.92, loss=0.355] Validation: 4: 100%| Epoch 5: 100%| 340/340 [00:06<00:00, 50.21 batch/s, accurracy=1, loss=0.0385] Validation: 5: 100%| 76/76 [00:00<00:00, 124.52 batch/s, accurracy=0.84, loss=0.412] 340/340 [00:06<00:00, 50.09 batch/s, accurracy=1, loss=0.021] Epoch 6: 100%| Validation: 6: 100%| 76/76 [00:00<00:00, 138.31 batch/s, accurracy=0.9, loss=0.227] 340/340 [00:06<00:00, 49.07 batch/s, accurracy=1, loss=0.225] Epoch 7: 100%| Validation: 7: 100%| 76/76 [00:00<00:00, 130.48 batch/s, accurracy=0.88, loss=0.282] 340/340 [00:06<00:00, 49.27 batch/s, accurracy=1, loss=0.119] Epoch 8: 100%| Validation: 8: 100%| 76/76 [00:00<00:00, 152.10 batch/s, accurracy=0.86, loss=0.351] Epoch 9: 100%| 340/340 [00:07<00:00, 48.11 batch/s, accurracy=1, loss=0.0589] 76/76 [00:00<00:00, 127.25 batch/s, accurracy=0.88, loss=0.314] Validation: 9: 100%| Epoch 10: 100%| 340/340 [00:06<00:00, 54.11 batch/s, accurracy=1, loss=0.0574] 76/76 [00:00<00:00, 154.88 batch/s, accurracy=0.8, loss=0.403] Validation: 10: 100%| Epoch 11: 100%| 340/340 [00:05<00:00, 56.80 batch/s, accurracy=1, loss=0.254] Validation: 11: 100%| 76/76 [00:00<00:00, 163.88 batch/s, accurracy=0.96, loss=0.216] Epoch 12: 100%| 340/340 [00:05<00:00, 57.47 batch/s, accurracy=1, loss=0.156] Validation: 12: 100%| 76/76 [00:00<00:00, 153.47 batch/s, accurracy=0.82, loss=0.487] Epoch 13: 100%| 340/340 [00:05<00:00, 57.83 batch/s, accurracy=1, loss=0.0658] Validation: 13: 100%| 76/76 [00:00<00:00, 154.07 batch/s, accurracy=0.92, loss=0.2] Epoch 14: 100%| 340/340 [00:05<00:00, 56.98 batch/s, accurracy=1, loss=0.0601] Validation: 14: 100%| 76/76 [00:00<00:00, 147.39 batch/s, accurracy=0.94, loss=0.237] Epoch 15: 100%| 340/340 [00:06<00:00, 53.63 batch/s, accurracy=1, loss=0.033] Validation: 15: 100%| 76/76 [00:00<00:00, 120.75 batch/s, accurracy=0.92, loss=0.215] Epoch 16: 100%| 340/340 [00:07<00:00, 47.01 batch/s, accurracy=1, loss=0.00694] Validation: 16: 100%| 76/76 [00:00<00:00, 138.89 batch/s, accurracy=0.86, loss=0.561] Epoch 17: 100%| 340/340 [00:06<00:00, 50.09 batch/s, accurracy=1, loss=0.0298] Validation: 17: 100%| 76/76 [00:00<00:00, 124.71 batch/s, accurracy=0.94, loss=0.0985] Epoch 18: 100%| 340/340 [00:06<00:00, 49.90 batch/s, accurracy=0.75, loss=0.295] Validation: 18: 100%| 76/76 [00:00<00:00, 139.97 batch/s, accurracy=0.86, loss=0.52] Epoch 19: 100%| 340/340 [00:06<00:00, 50.88 batch/s, accurracy=1, loss=0.033] Validation: 19: 100%| 76/76 [00:00<00:00, 116.70 batch/s, accurracy=0.88, loss=0.376] Epoch 20: 100%| 340/340 [00:06<00:00, 49.02 batch/s, accurracy=1, loss=0.0197] Validation: 20: 100%| 76/76 [00:00<00:00, 116.72 batch/s, accurracy=0.98, loss=0.0836] Epoch 21: 100%| 340/340 [00:06<00:00, 50.68 batch/s, accurracy=1, loss=0.075] Validation: 21: 100%| 76/76 [00:00<00:00, 131.27 batch/s, accurracy=0.9, loss=0.379] Epoch 22: 100%| 340/340 [00:06<00:00, 50.23 batch/s, accurracy=1, loss=0.0487] Validation: 22: 100%| 76/76 [00:00<00:00, 125.02 batch/s, accurracy=0.84, loss=0.369] Epoch 23: 100%| 340/340 [00:06<00:00, 51.22 batch/s, accurracy=1, loss=0.0527] Validation: 23: 100%| 76/76 [00:00<00:00, 136.93 batch/s, accurracy=0.94, loss=0.227] Epoch 24: 100%| 340/340 [00:06<00:00, 51.04 batch/s, accurracy=1, loss=0.0666] Validation: 24: 100%| 76/76 [00:00<00:00, 132.07 batch/s, accurracy=0.96, loss=0.583] Epoch 25: 100%| 340/340 [00:06<00:00, 51.19 batch/s, accurracy=1, loss=0.0122] Validation: 25: 100%| 76/76 [00:00<00:00, 138.55 batch/s, accurracy=0.94, loss=0.159] Epoch 26: 100%| 340/340 [00:06<00:00, 48.99 batch/s, accurracy=1, loss=0.0256] Validation: 26: 100%| 76/76 [00:00<00:00, 146.68 batch/s, accurracy=0.86, loss=0.325] Epoch 27: 100%| 340/340 [00:06<00:00, 51.50 batch/s, accurracy=1, loss=0.0169] Validation: 27: 100%| 76/76 [00:00<00:00, 137.55 batch/s, accurracy=0.88, loss=0.251] Epoch 28: 100%| 340/340 [00:06<00:00, 50.05 batch/s, accurracy=0.75, loss=1.6] Validation: 28: 100%| 76/76 [00:00<00:00, 136.20 batch/s, accurracy=0.82, loss=0.464] Epoch 29: 100%| 340/340 [00:06<00:00, 49.98 batch/s, accurracy=1, loss=0.0767] Validation: 29: 100%| 76/76 [00:00<00:00, 123.22 batch/s, accurracy=0.84, loss=0.693] Epoch 30: 100%| 340/340 [00:07<00:00, 48.35 batch/s, accurracy=1, loss=0.0422] Validation: 30: 100%| 76/76 [00:00<00:00, 132.65 batch/s, accurracy=0.96, loss=0.215] Epoch 31: 100%| 340/340 [00:06<00:00, 50.11 batch/s, accurracy=1, loss=0.0102] Validation: 31: 100%| 76/76 [00:00<00:00, 134.87 batch/s, accurracy=0.9, loss=0.206] Epoch 32: 100%| 340/340 [00:06<00:00, 50.93 batch/s, accurracy=1, loss=0.154] Validation: 32: 100%| 76/76 [00:00<00:00, 142.81 batch/s, accurracy=0.94, loss=0.318] Epoch 33: 100%| 340/340 [00:06<00:00, 50.37 batch/s, accurracy=1, loss=0.0167] Validation: 33: 100%| 76/76 [00:00<00:00, 133.03 batch/s, accurracy=0.9, loss=0.421] Epoch 34: 100%| 340/340 [00:06<00:00, 50.19 batch/s, accurracy=1, loss=0.0273] 76/76 [00:00<00:00, 139.28 batch/s, accurracy=0.92, loss=0.194] Validation: 34: 100%| Epoch 35: 100%| 340/340 [00:06<00:00, 50.29 batch/s, accurracy=1, loss=0.00164] Validation: 35: 100%| 76/76 [00:00<00:00, 128.29 batch/s, accurracy=0.9, loss=0.422] Epoch 36: 100%| 340/340 [00:06<00:00, 49.49 batch/s, accurracy=0.75, loss=0.474] Validation: 36: 100%| 76/76 [00:00<00:00, 140.86 batch/s, accurracy=0.9, loss=0.539] Epoch 37: 100%| 340/340 [00:06<00:00, 50.29 batch/s, accurracy=1, loss=0.125] Validation: 37: 100%| 76/76 [00:00<00:00, 142.97 batch/s, accurracy=0.9, loss=0.23] Epoch 38: 100%| 340/340 [00:06<00:00, 51.02 batch/s, accurracy=1, loss=0.0303] Validation: 38: 100%| 76/76 [00:00<00:00, 125.85 batch/s, accurracy=0.96, loss=0.105] Epoch 39: 100%| 340/340 [00:06<00:00, 49.23 batch/s, accurracy=1, loss=0.0738] 76/76 [00:00<00:00, 130.64 batch/s, accurracy=0.92, loss=0.452] Validation: 39: 100%| Epoch 40: 100%| 340/340 [00:06<00:00, 50.23 batch/s, accurracy=1, loss=0.0343] Validation: 40: 100% 76/76 [00:00<00:00, 126.79 batch/s, accurracy=0.9, loss=0.202] 340/340 [00:06<00:00, 49.61 batch/s, accurracy=1, loss=0.0327] Epoch 41: 100%| 76/76 [00:00<00:00, 124.90 batch/s, accurracy=0.96, loss=0.18] Validation: 41: 100%| Epoch 42: 100%| | 340/340 [00:06<00:00, 49.66 batch/s, accurracy=1, loss=0.0553] | 76/76 [00:00<00:00, 138.55 batch/s, accurracy=0.94, loss=0.202] Validation: 42: 100%| | 340/340 [00:06<00:00, 49.40 batch/s, accurracy=1, loss=0.022] Epoch 43: 100%| | 76/76 [00:00<00:00, 144.19 batch/s, accurracy=0.88, loss=0.432] Validation: 43: 100%| Epoch 44: 100%| | 340/340 [00:06<00:00, 51.24 batch/s, accurracy=1, loss=0.0275] | 76/76 [00:00<00:00, 118.50 batch/s, accurracy=0.96, loss=0.157] Validation: 44: 100%| | 340/340 [00:07<00:00, 48.55 batch/s, accurracy=1, loss=0.0225] Epoch 45: 100%| 76/76 [00:00<00:00, 117.10 batch/s, accurracy=0.96, loss=0.0771] Validation: 45: 100%| Epoch 46: 100%| 340/340 [00:06<00:00, 49.09 batch/s, accurracy=1, loss=0.0735] Validation: 46: 100%| 76/76 [00:00<00:00, 137.30 batch/s, accurracy=0.96, loss=0.105] Epoch 47: 100%| | 340/340 [00:07<00:00, 47.97 batch/s, accurracy=1, loss=0.0176] | 76/76 [00:00<00:00, 133.75 batch/s, accurracy=0.88, loss=0.795] Validation: 47: 100%| | 340/340 [00:07<00:00, 48.42 batch/s, accurracy=1, loss=0.00247] Epoch 48: 100%| Validation: 48: 100%| 76/76 [00:00<00:00, 121.33 batch/s, accurracy=0.96, loss=0.261] 340/340 [00:07<00:00, 47.34 batch/s, accurracy=1, loss=0.00522] Epoch 49: 100%| Validation: 49: 100%| | 76/76 [00:00<00:00, 127.11 batch/s, accurracy=0.92, loss=0.217] Generate Metrics on the Dev Set Here we recreate the dev dataloader to go a single document at a time. We use the less robust predict method so we can explicitly show the values and calculations being performed. In [6]: dev dataloader = datasets.create torch dataloader(val dataset, vocab, label transform, text transform, weighte model.eval() preds = [] labels = [] for batch in dev\_dataloader: label, text, \*\_ = batch pred\_label = train.predict(model, text) preds.append(pred label) labels.append(label.cpu().item()) Generate a Confusion Matrix We can use the confusion matrix to make it easy to visualize the values for the metrics. P0 P1 A0 TN FP A1 FN TP In [7]: from sklearn.metrics import confusion matrix, average precision score cm = confusion matrix(labels, preds) print(cm) [[4352 348] [ 94 56]] Calculate Precision, Recall, F1-Score on Dev Set We use a confusion matrix to make it easy to map out the values for true positive, true negative, false positive, and false negative. • Precision = tp / (tp + fp) • Recall = tp / (tp + fn)• F1 Score = 2 (precision recall) / (precision + recall) In [8]: tn, fp, fn, tp = cm.ravel() # Extract the components # Calculate and print Precision precision string =  $f''\{tp\} / (\{tp\} + \{fp\})''$ precision = round(eval(precision string), 4) print(f"Precision: {precision\_string} = {precision}") # Calculate and print Recall recall\_string =  $f''\{tp\} / (\{tp\} + \{fn\})''$ recall = round(eval(recall string), 4) print(f"Recall: {recall\_string} = {recall}") # Calculate and print F1 Score f1 string = f"2 \* ({precision} \* {recall}) / ({precision} + {recall})" f1 = eval(f1 string) $print(f"F1: {f1 string} = {round(f1, 4)}")$ print(f"AP: {average precision score(labels, preds)}") Precision: 56 / (56 + 348) = 0.1386Recall: 56 / (56 + 94) = 0.3733F1: 2 \* (0.1386 \* 0.3733) / (0.1386 + 0.3733) = 0.2021AP: 0.07113061821646083 **Update Methods** Because of the poor performance, we are going to update the vocab and model. • Use GloVe embeddings within the model as well as the basis for the vocabulary. • Update the model to be a 3-layer MLP with Activation to introduce non-linearity. Repeat Experiment with "Title", "Abstract", and "Keyword" Data We will keep everything exactly the same for setup, changing only the things needed. In [9]: use\_columns = ["title", "abstract", "keywords"] data\_columns = [datafield\_map[col] for col in use\_columns] ngrams = 1glove = ir vocab.create glove with unk vector() vocab = ir vocab.create vocab from glove(glove) # Load the TSV into datasets with the appropriate feature columns. train dataset = datasets.TSVRawTextMapDataset("../datasets/systematic\_review/phase1.train.shuf.tsv", data\_colum val dataset = datasets.TSVRawTextMapDataset("../datasets/systematic review/phase1.dev.shuf.tsv", data columns) # Instantiate the dataloaders. train\_dataloader = datasets.create\_torch\_dataloader(train\_dataset, vocab, label\_transform, text\_transform, wei val\_dataloader = datasets.create\_torch\_dataloader(val\_dataset, vocab, label\_transform, text\_transform, weighte # Create Model model = models.EmbeddingBagMLPModel(num class=num classes, hidden layer size=100, embedding vectors=glove.vector In [12]: EPOCHS = 50learning rate = 0.001 # Create the loss function weighted to inverse class distribution loss function = torch.nn.CrossEntropyLoss() # Instantiate a Stochastic Gradient Descent optimizer and "Auto" Learning Rate schedule. optimizer = torch.optim.Adam(model.parameters(), lr=learning rate) scheduler = torch.optim.lr scheduler.StepLR(optimizer, 1.0, gamma=0.95) # Tensorboard writing utility class. writer = SummaryWriter("runs/TAK MLP") # Perform Training for i in range(EPOCHS): start iter = len(train dataloader) \* i train.train epoch(i, model, optimizer, loss function, train dataloader, start iter=start iter, writer=write validation results = train.evaluate epoch(i, model, loss function, val dataloader, writer) scheduler.step() torch.save(model.state dict(), "model weights/TAK MLP state dict.pth") Epoch 0: 100% | 340/340 [00:14<00:00, 23.62 batch/s, accurracy=0.75, loss=0.994] 76/76 [00:02<00:00, 29.22 batch/s, accurracy=0.78, loss=0.357] Validation: 0: 100%| Epoch 1: 100%| 340/340 [00:13<00:00, 24.55 batch/s, accurracy=0.75, loss=0.299] Validation: 1: 100%| 76/76 [00:02<00:00, 29.60 batch/s, accurracy=0.9, loss=0.218] Epoch 2: 100%| 340/340 [00:13<00:00, 24.98 batch/s, accurracy=0.25, loss=1.29] Validation: 2: 100%| 76/76 [00:02<00:00, 28.30 batch/s, accurracy=0.76, loss=0.519] Epoch 3: 100%| | 340/340 [00:13<00:00, 25.43 batch/s, accurracy=1, loss=0.179] Validation: 3: 100%| | 76/76 [00:02<00:00, 25.76 batch/s, accurracy=0.84, loss=0.283] 340/340 [00:13<00:00, 26.04 batch/s, accurracy=0.75, loss=0.538] Epoch 4: 100%| Validation: 4: 100%| | 76/76 [00:02<00:00, 29.29 batch/s, accurracy=0.84, loss=0.319] Epoch 5: 100%| | 340/340 [00:12<00:00, 26.30 batch/s, accurracy=0.75, loss=0.337] Validation: 5: 100%| 76/76 [00:02<00:00, 29.53 batch/s, accurracy=0.78, loss=0.36] Epoch 6: 100%| | 340/340 [00:12<00:00, 26.23 batch/s, accurracy=1, loss=0.0418] Validation: 6: 100%| | 76/76 [00:02<00:00, 28.43 batch/s, accurracy=0.86, loss=0.319] Epoch 7: 100%| | 340/340 [00:13<00:00, 25.90 batch/s, accurracy=1, loss=0.137] 76/76 [00:02<00:00, 27.00 batch/s, accurracy=0.84, loss=0.294] Validation: 7: 100%| Epoch 8: 100%| 340/340 [00:13<00:00, 25.74 batch/s, accurracy=0.75, loss=0.296] Validation: 8: 100%| 76/76 [00:02<00:00, 29.07 batch/s, accurracy=0.82, loss=0.328] Epoch 9: 100%| 340/340 [00:12<00:00, 26.52 batch/s, accurracy=0.75, loss=0.298] Validation: 9: 100%| 76/76 [00:02<00:00, 29.01 batch/s, accurracy=0.84, loss=0.322] Epoch 10: 100%| 340/340 [00:13<00:00, 25.97 batch/s, accurracy=1, loss=0.0966] Validation: 10: 100%| 76/76 [00:02<00:00, 27.86 batch/s, accurracy=0.74, loss=0.5] Epoch 11: 100%| 340/340 [00:13<00:00, 25.77 batch/s, accurracy=1, loss=0.0825] Validation: 11: 100%| 76/76 [00:02<00:00, 28.20 batch/s, accurracy=0.92, loss=0.196] Epoch 12: 100%| 340/340 [00:13<00:00, 25.55 batch/s, accurracy=0.75, loss=0.367] Validation: 12: 100%| 76/76 [00:02<00:00, 25.81 batch/s, accurracy=0.86, loss=0.205] Epoch 13: 100%| 340/340 [00:13<00:00, 25.51 batch/s, accurracy=1, loss=0.129] Validation: 13: 100%| 76/76 [00:02<00:00, 27.89 batch/s, accurracy=0.82, loss=0.349] Epoch 14: 100%| 340/340 [00:13<00:00, 25.17 batch/s, accurracy=1, loss=0.134] Validation: 14: 100%| 76/76 [00:02<00:00, 26.59 batch/s, accurracy=0.82, loss=0.279] | 340/340 [00:13<00:00, 25.09 batch/s, accurracy=0.75, loss=0.419] Epoch 15: 100%| Validation: 15: 100%| 76/76 [00:03<00:00, 22.45 batch/s, accurracy=0.86, loss=0.304] Epoch 16: 100%| 340/340 [00:15<00:00, 22.66 batch/s, accurracy=1, loss=0.00779] Validation: 16: 100%| 76/76 [00:03<00:00, 22.72 batch/s, accurracy=0.82, loss=0.282] | 340/340 [00:15<00:00, 22.38 batch/s, accurracy=1, loss=0.125] Epoch 17: 100%| Validation: 17: 100%| 76/76 [00:03<00:00, 23.23 batch/s, accurracy=0.8, loss=0.409] Epoch 18: 100%| 340/340 [00:15<00:00, 21.93 batch/s, accurracy=0.75, loss=0.758] Validation: 18: 100%| 76/76 [00:03<00:00, 24.47 batch/s, accurracy=0.82, loss=0.38] Epoch 19: 100%| 340/340 [00:15<00:00, 22.55 batch/s, accurracy=1, loss=0.168] Validation: 19: 100%| 76/76 [00:03<00:00, 24.59 batch/s, accurracy=0.88, loss=0.263] Epoch 20: 100%| 340/340 [00:14<00:00, 23.27 batch/s, accurracy=0.75, loss=0.746] Validation: 20: 100%| 76/76 [00:03<00:00, 24.57 batch/s, accurracy=0.82, loss=0.306] Epoch 21: 100%| 340/340 [00:14<00:00, 22.98 batch/s, accurracy=0.75, loss=0.327] Validation: 21: 100%| 76/76 [00:02<00:00, 26.71 batch/s, accurracy=0.94, loss=0.211] Epoch 22: 100%| | 340/340 [00:13<00:00, 24.95 batch/s, accurracy=1, loss=0.0574] Validation: 22: 100%| 76/76 [00:02<00:00, 28.22 batch/s, accurracy=0.8, loss=0.344] Epoch 23: 100%| 340/340 [00:13<00:00, 25.97 batch/s, accurracy=1, loss=0.0406] Validation: 23: 100%| 76/76 [00:02<00:00, 25.36 batch/s, accurracy=0.88, loss=0.209] Epoch 24: 100%| 340/340 [00:14<00:00, 23.73 batch/s, accurracy=1, loss=0.0563] Validation: 24: 100%| | 76/76 [00:02<00:00, 25.59 batch/s, accurracy=0.9, loss=0.143] Epoch 25: 100%| 340/340 [00:13<00:00, 25.84 batch/s, accurracy=0.75, loss=0.245] Validation: 25: 100%| 76/76 [00:02<00:00, 27.95 batch/s, accurracy=0.9, loss=0.157] Epoch 26: 100%| 340/340 [00:13<00:00, 25.51 batch/s, accurracy=1, loss=0.153] Validation: 26: 100%| 76/76 [00:02<00:00, 27.60 batch/s, accurracy=0.92, loss=0.188] Epoch 27: 100%| 340/340 [00:13<00:00, 25.56 batch/s, accurracy=1, loss=0.0932] Validation: 27: 100%| 76/76 [00:02<00:00, 27.51 batch/s, accurracy=0.96, loss=0.104] Epoch 28: 100%| 340/340 [00:13<00:00, 25.37 batch/s, accurracy=1, loss=0.0429] Validation: 28: 100%| 76/76 [00:02<00:00, 27.24 batch/s, accurracy=0.8, loss=0.439] Epoch 29: 100%| 340/340 [00:13<00:00, 25.26 batch/s, accurracy=1, loss=0.0793] Validation: 29: 100%| 76/76 [00:02<00:00, 28.76 batch/s, accurracy=0.88, loss=0.239] 340/340 [00:13<00:00, 25.40 batch/s, accurracy=1, loss=0.117] Epoch 30: 100%| Validation: 30: 100%| | 76/76 [00:02<00:00, 28.14 batch/s, accurracy=0.92, loss=0.136] Epoch 31: 100%| | 340/340 [00:13<00:00, 25.16 batch/s, accurracy=0.75, loss=0.479] 76/76 [00:02<00:00, 28.87 batch/s, accurracy=0.9, loss=0.213] Validation: 31: 100%| Epoch 32: 100%| | 340/340 [00:13<00:00, 25.64 batch/s, accurracy=1, loss=0.038] Validation: 32: 100%| | 76/76 [00:03<00:00, 22.90 batch/s, accurracy=0.88, loss=0.298] Epoch 33: 100%| | 340/340 [00:14<00:00, 24.07 batch/s, accurracy=0.75, loss=0.246] Validation: 33: 100%| | 76/76 [00:02<00:00, 27.83 batch/s, accurracy=0.82, loss=0.457] Epoch 34: 100%| | 340/340 [00:13<00:00, 24.65 batch/s, accurracy=1, loss=0.0358] 76/76 [00:03<00:00, 24.36 batch/s, accurracy=0.86, loss=0.226] Validation: 34: 100%| Epoch 35: 100%| | 340/340 [00:14<00:00, 24.02 batch/s, accurracy=1, loss=0.0942] Validation: 35: 100%| | 76/76 [00:02<00:00, 27.15 batch/s, accurracy=0.9, loss=0.174] Epoch 36: 100%| 340/340 [00:13<00:00, 25.65 batch/s, accurracy=1, loss=0.0324] Validation: 36: 100%| | 76/76 [00:02<00:00, 26.43 batch/s, accurracy=0.8, loss=0.427] Epoch 37: 100%| 340/340 [00:13<00:00, 25.20 batch/s, accurracy=1, loss=0.27] 76/76 [00:02<00:00, 26.31 batch/s, accurracy=0.9, loss=0.165] Validation: 37: 100%| 340/340 [00:13<00:00, 25.65 batch/s, accurracy=1, loss=0.0189] Epoch 38: 100%| Validation: 38: 100% | 76/76 [00:02<00:00, 27.20 batch/s, accurracy=0.9, loss=0.299] Epoch 39: 100%| | 340/340 [00:13<00:00, 25.26 batch/s, accurracy=1, loss=0.0542] Validation: 39: 100%| | 76/76 [00:02<00:00, 26.00 batch/s, accurracy=0.86, loss=0.213] Epoch 40: 100%| 340/340 [00:13<00:00, 25.22 batch/s, accurracy=1, loss=0.0553] Validation: 40: 100%| 76/76 [00:03<00:00, 25.12 batch/s, accurracy=0.78, loss=0.426] Epoch 41: 100%| | 340/340 [00:13<00:00, 25.00 batch/s, accurracy=1, loss=0.064] | 76/76 [00:02<00:00, 26.98 batch/s, accurracy=0.9, loss=0.192] Validation: 41: 100%| Epoch 42: 100%| | 340/340 [00:13<00:00, 25.01 batch/s, accurracy=1, loss=0.0433] 76/76 [00:02<00:00, 27.93 batch/s, accurracy=0.94, loss=0.127] Validation: 42: 100%| Epoch 43: 100%| | 340/340 [00:13<00:00, 24.92 batch/s, accurracy=0.75, loss=0.789] Validation: 43: 100%| 76/76 [00:02<00:00, 27.59 batch/s, accurracy=0.82, loss=0.32] Epoch 44: 100%| | 340/340 [00:13<00:00, 25.21 batch/s, accurracy=0.75, loss=0.518] 76/76 [00:02<00:00, 26.37 batch/s, accurracy=0.82, loss=0.291] Validation: 44: 100%| Epoch 45: 100%| 340/340 [00:13<00:00, 24.77 batch/s, accurracy=0.75, loss=0.361] Validation: 45: 100%| 76/76 [00:02<00:00, 27.78 batch/s, accurracy=0.86, loss=0.279] Epoch 46: 100%| 340/340 [00:15<00:00, 22.41 batch/s, accurracy=1, loss=0.0211] Validation: 46: 100%| | 76/76 [00:02<00:00, 26.13 batch/s, accurracy=0.92, loss=0.204] Epoch 47: 100%| 340/340 [00:14<00:00, 23.33 batch/s, accurracy=0.75, loss=0.401] Validation: 47: 100%| | 76/76 [00:02<00:00, 25.71 batch/s, accurracy=0.96, loss=0.104] Epoch 48: 100%| | 340/340 [00:13<00:00, 25.33 batch/s, accurracy=1, loss=0.0586] Validation: 48: 100%| | 76/76 [00:03<00:00, 25.10 batch/s, accurracy=0.9, loss=0.194] Epoch 49: 100% | 340/340 [00:13<00:00, 25.73 batch/s, accurracy=1, loss=0.0563] Validation: 49: 100%| 76/76 [00:03<00:00, 20.82 batch/s, accurracy=0.88, loss=0.24] In [13]: dev dataloader = datasets.create torch dataloader(val dataset, vocab, label transform, text transform, weighte model.eval() preds = [] labels = []for batch in dev dataloader: label, text, \* = batch pred label = train.predict(model, text) preds.append(pred label) labels.append(label.cpu().item()) cm = confusion matrix(labels, preds) print(cm) tn, fp, fn, tp = cm.ravel() # Extract the components # Calculate and print Precision precision string =  $f''\{tp\} / (\{tp\} + \{fp\})''$ precision = round(eval(precision string), 4) print(f"Precision: {precision string} = {precision}") # Calculate and print Recall recall string =  $f''\{tp\} / (\{tp\} + \{fn\})''$ recall = round(eval(recall string), 4) print(f"Recall: {recall string} = {recall}") # Calculate and print F1 Score f1 string = f"2 \* ({precision} \* {recall}) / ({precision} + {recall})" f1 = eval(f1 string)  $print(f"F1: {f1 string} = {round(f1, 4)}")$ print(f"AP: {average precision score(labels, preds)}") [[4120 580] [ 24 126]] Precision: 126 / (126 + 580) = 0.1785Recall: 126 / (126 + 24) = 0.84F1: 2 \* (0.1785 \* 0.84) / (0.1785 + 0.84) = 0.2944AP: 0.15486346777255336 In [15]: %load\_ext tensorboard %tensorboard --logdir runs/TAK MLP **Generate Predictions On Test Data** In [29]: test dataset = datasets.TSVRawTextMapDataset("../datasets/systematic review/phase1.test.shuf.tsv", data columns test dataloader = datasets.create torch dataloader(test dataset, vocab, label transform, text transform, weight model.eval() preds = [] doc ids = []with open("RCERESA1.txt", "w") as f: for batch in test dataloader: label, text, offsets, doc id = batch pred label = train.predict(model, text) pred = pred label if pred label == 1 else -1 line =  $f''\{doc id[0]\}\t\{pred\}\n''$ f.write(line)

clas	Datum
clas	<pre>Returns:     A PyTorch DataLoader to be used during training, eval, or test. """  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  def _collate_batch(batch):     label_list, docid_list, text_list, offsets = [], [], [], [0]     for (_label, _docid, _text) in batch:         label_list.append(label_transform(_label))         processed_text = torch.tensor(</pre>
clas	<pre>processed_text = torch.tensor(</pre>
,	<pre>if weighted:     weights = dataset.sample_weights     sampler = WeightedRandomSampler(weights=weights, num_samples=len(weights)) else:     sampler = None  return data.DataLoader(     dataset,     collate_fn=_collate_batch,     shuffle=(sampler is None),</pre>
	<pre>sampler=sampler,    **kwargs )  s TSVRawTextIterableDataset(data.IterableDataset): """Dataset that loads TSV data incrementally as an iterable and returns raw text. This dataset must be traversed in order as it only reads from the TSV file as it is called.</pre>
	Useful if the size of data is too large to load into memory at once.  """  definit(self, filepath: str, data_columns: List[int]):     """Loads an iterator from a file.  Args:     filepath: location of the .tsv file     data_columns: the columns in the .tsv that are used as feature data  """
,	<pre>selfnumber_of_items = _get_tsv_file_length(filepath) selfiterator = _create_data_from_tsv(         filepath, data_column_indices=data_columns ) selfcurrent_position = 0  defiter(self):     return self</pre>
	<pre>defnext(self):     item = next(selfiterator)     selfcurrent_position += 1     return item  deflen(self):     return selfnumber_of_items  s TSVRawTextMapDataset(data.Dataset):</pre>
,	<pre>"""Dataset that loads all TSV data into memory and returns raw text.  This dataset provides a map interface, allowing access to any entry. Useful for modifying the sampling or order during training. """  definit(self, filepath: str, data_columns: List[int]):     """Loads .tsv structed data into memory.</pre>
	<pre>Args:     filepath: location of the .tsv file     data_columns: the columns in the .tsv that are used as feature data """  selfrecords = list(     _create_data_from_tsv(filepath, data_column_indices=data_columns) ) selfsample_weights, selfclass_weights = selfcalculate_weights()</pre>
	<pre>@property def sample_weights(self):     return selfsample_weights  @property def class_weights(self):     return selfclass_weights  def _calculate_weights(self):     targets = torch.tensor(</pre>
	<pre>[label if label &gt; 0 else 0 for label, *_ in selfrecords] ) unique, sample_counts = torch.unique(targets, return_counts=True) weight = 1.0 / sample_counts sample_weights = torch.tensor([weight[t] for t in targets]) class_weights = weight / weight.sum() return sample_weights, class_weights  defgetitem(self, index):</pre>
lef .	<pre>return selfrecords[index]  deflen(self):     return len(selfrecords)  _create_data_from_tsv(data_path, data_column_indices): with io.open(data_path, encoding="utf8") as f:     reader = unicode_csv_reader(f, delimiter="\t")</pre>
,	<pre>for row in reader:     data = [row[i] for i in data_column_indices]     yield int(row[0]), row[1], " ".join(data)  _get_tsv_file_length(data_path): with io.open(data_path, encoding="utf8") as f:     row_count = sum(1 for row in f)</pre> return row_count
impo From impo impo From	<pre>rt io   typing import List rt numpy as np rt torch   torchtext.data.utils import get_tokenizer, ngrams_iterator   torchtext.utils import unicode_csv_reader</pre>
rom rom	<pre>torchtext.vocab import build_vocab_from_iterator torchtext.vocab import vocab as vocab_builder torchtext.vocab import GloVe  create_glove_with_unk_vector() -&gt; GloVe: glove = GloVe() # Load the average vector for this glove embedding set to use for defaults. average_glove_vector = np.load("/datasets/glove_default_vector.npy") unk_init_vec_=_torch_from_numpy(avenage_glove_vector)</pre>
lef	<pre>unk_init_vec = torch.from_numpy(average_glove_vector) # Extend the glove vectors with one for "unk" glove.vectors = torch.cat((glove.vectors, unk_init_vec.unsqueeze(0)))  return glove  create_vocab_from_glove(glove: GloVe): # Since glove is already ordered and not a counter, we overload the # Constructor to align the indices. unk_token = "<unk>"</unk></pre>
lef	<pre>vocab = vocab_builder(glove.stoi, min_freq=0) vocab.append_token(unk_token) vocab.set_default_index(vocab[unk_token])  return vocab  create_vocab_from_tsv( filepath: str, column_indices_to_use: List[int],</pre>
:	<pre>minimum_word_freq: int = 1, ngrams: int = 1, """Creates a PyTorch vocab object from a TSV file. The resulting vocab object converts words to indices for assisting in embedding and DL operations. Args:     filepath: The location of the TSV file</pre>
	<pre>minimum_word_freq: How many times a word must appear to be included    ngrams: The size of ngrams to use for the vocab    column_indices_to_use: Which columns from the TSV are part of the actual feature set  Returns:    A torchtext vocab object. """ unk_token = "<unk>" vocab = build_vocab_from_iterator(    _tsv_iterator(filepath, ngrams=ngrams, column_indices=column_indices_to_use),</unk></pre>
lef .	<pre>min_freq=minimum_word_freq,     specials=[unk_token], ) vocab.set_default_index(vocab[unk_token]) return vocab  _tsv_iterator(data_path, ngrams, column_indices): # Spacy has novel tokenizer</pre>
,	<pre>tokenizer = get_tokenizer("basic_english") with io.open(data_path, encoding="utf8") as f:     reader = unicode_csv_reader(f, delimiter="\t")     for row in reader:         row_iter = [row[i] for i in column_indices]         tokens = " ".join(row_iter)         yield ngrams_iterator(tokenizer(tokens), ngrams)</pre>
rom rom mpo mpo rom	<pre>collections import Counter logging import log typing import Any, Callable, Dict, Tuple  rt torch rt torch.nn as nn sklearn.metrics import precision_recall_fscore_support, average_precision_score torch.utils import data</pre>
rom rom	torch.utils.tensorboard import SummaryWriter tqdm import tqdm  predict(model: nn.Module, text: torch.Tensor) -> int: """Predicts the class of a specific text given converted features.  Args:     model: the model to use for prediction/inferrence
I	<pre>model: the model to use for prediction/inferrence     text: the previously converted text (using the prior dictionary)  Returns:     The predicted label for the provided text. """ model.eval() no_offset = torch.tensor([0]) with torch.no_grad():     pred_scores = model(text, no_offset)     pred_label = pred_scores.argmax(1).item()</pre>
	<pre>return pred_label  train_epoch( epoch_num: int, model: nn.Module, optimizer: torch.optim.Optimizer, loss_function: Callable,</pre>
->	<pre>dataloader: data.DataLoader, start_iter: int = 0, log_interval: int = 100, writer: SummaryWriter = None,   int: """Performs training on a single pass through a dataloader.  Args:     epoch_num: The current number of this epoch of training.     model: The PyTorch module to train</pre>
	optimizer: The optimizer to use for training. loss_function: The function that calculates loss between truth and prediction. dataloader: Provides a properly formatted batch of data at each iteration. start_iter: The iteration this epoch started on. Used for plotting. log_interval: How often to log the scores. writier: Tensorboard summary writer.  Returns: The value of the start_iter plus number of batches performed this epoch.
	<pre>batch_counter = start_iter model.train() with tqdm(dataloader, unit=" batch", bar_format="{desc:&gt;20}{percentage:3.0f}% {bar}{r_bar}") as tepe for batch in tepoch:     batch_counter += 1     tepoch.set_description(f"Epoch {epoch_num}")     results = train_step(batch, model, optimizer, loss_function)     tepoch.set_postfix(loss=results["loss"], accurracy=results["accuracy"])</pre>
lef	<pre>if writer is not None and batch_counter % log_interval == 0:</pre>
->	<pre>optimizer: torch.optim.Optimizer, loss_function: Callable, Dict[str, float]: """Performs a single training step on a model.  Args:     batch: A previously formatted batch of data.     model: Torch model to perform training on.     optimizer: The optmizer class used in training     loss_function: The callable function to generate loss between prediction and truth.</pre>
	Returns:  The different metrics generated this training step.  """  labels, text, offsets, *_ = batch  optimizer.zero_grad()  predicted_scores = model(text, offsets)  loss = loss_function(predicted_scores, labels)
	<pre>loss.backward() torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1) optimizer.step()  predicted_labels = predicted_scores.argmax(1)  accuracy = (predicted_labels == labels).sum().item() / labels.size(0)  y_true = labels.detach().cpu().numpy()</pre>
	<pre>y_pred = predicted_labels.cpu().numpy() precision, recall, fscore, support = precision_recall_fscore_support(     y_true,     y_pred,     average="binary",     zero_division=0, )  results = {     "loss": loss.detach().cpu().item(),</pre>
	"accuracy": accuracy,     "precision": precision,     "recall": recall,     "fscore": fscore, } return results  evaluate_epoch(
)->	<pre>epoch_num: int, model: nn.Module, loss_function: Callable, dataloader: data.DataLoader, writer: SummaryWriter = None, Dict[str, float]: """Performs validation on a single pass through a dataloader.  Args:     epoch_num: The current number of this epoch of training.</pre>
	<pre>model: The PyTorch module to train   loss_function: The function that calculates loss between truth and prediction.   dataloader: Provides a properly formatted batch of data at each iteration.   writier: Tensorboard summary writer.  Returns:   The average validation metrics for the whole dataset. """ model.eval()</pre>
	<pre>aggregate_results = Counter() with tqdm(dataloader, unit=" batch", bar_format="{desc:&gt;20}{percentage:3.0f}% {bar}{r_bar}") as tepo for batch in tepoch:     tepoch.set_description(f"Validation: {epoch_num}")     results = evaluate_step(batch, model, loss_function)     tepoch.set_postfix(loss=results["loss"], accurracy=results["accuracy"])     aggregate_results += Counter(results)  average_results = {     key: aggregate_results[key] / tepoch.total for key in aggregate_results</pre>
lef	<pre>writer.add_scalars("validation", average_results, epoch_num) return average_results  evaluate_step( batch: Tuple[torch.Tensor,], model: nn.Module,</pre>
) ->	<pre>loss_function: Callable, Dict[str, float]: """Performs a single validation step on a model.  Args:     batch: A previously formatted batch of data.     model: Torch model to perform training on.     loss_function: The callable function to generate loss between prediction and truth.</pre> Returns:
	<pre>The different metrics generated this training step. """  labels, text, offsets, *_ = batch  with torch.no_grad():     predicted_scores = model(text, offsets)     loss = loss_function(predicted_scores, labels)     predicted_labels = predicted_scores.argmax(1)     accuracy = (predicted_labels == labels).sum().item() / labels.size(0)</pre>
	<pre>y_true = labels.detach().cpu().numpy() y_pred = predicted_labels.cpu().numpy()  precision, recall, fscore, support = precision_recall_fscore_support(     y_true,     y_pred,     average="binary",     zero_division=0,</pre>
	<pre>results = {     "loss": loss.detach().cpu().item(),     "accuracy": accuracy,     "precision": precision,     "recall": recall,     "fscore": fscore, } return results</pre>
impo From impo impo From	rt io typing import List rt numpy as np rt torch torchtext.data.utils import get_tokenizer, ngrams_iterator torchtext.utils import unicode_csv_reader
From From	<pre>torchtext.vocab import build_vocab_from_iterator torchtext.vocab import vocab as vocab_builder torchtext.vocab import GloVe  create_glove_with_unk_vector() -&gt; GloVe: glove = GloVe() # Load the average vector for this glove embedding set to use for defaults. average_glove_vector = np.load("/datasets/glove_default_vector.npy")</pre>
lef	<pre>unk_init_vec = torch.from_numpy(average_glove_vector) # Extend the glove vectors with one for "unk" glove.vectors = torch.cat((glove.vectors, unk_init_vec.unsqueeze(0)))  return glove  create_vocab_from_glove(glove: GloVe): # Since glove is already ordered and not a counter, we overload the # Constructor to align the indices.</pre>
lef	<pre>unk_token = "<unk>" vocab = vocab_builder(glove.stoi, min_freq=0) vocab.append_token(unk_token) vocab.set_default_index(vocab[unk_token])  return vocab  create_vocab_from_tsv( filepath: str, column_indices_to_use: List[int],</unk></pre>
):	<pre>minimum_word_freq: int = 1, ngrams: int = 1, """Creates a PyTorch vocab object from a TSV file. The resulting vocab object converts words to indices for assisting in embedding and DL operations. Args:     filepath: The location of the TSV file</pre>
	<pre>minimum_word_freq: How many times a word must appear to be included    ngrams: The size of ngrams to use for the vocab    column_indices_to_use: Which columns from the TSV are part of the actual feature set  Returns:    A torchtext vocab object. """ unk_token = "<unk>" vocab = build_vocab_from_iterator(</unk></pre>
lef .	<pre>_tsv_iterator(filepath, ngrams=ngrams, column_indices=column_indices_to_use),     min_freq=minimum_word_freq,     specials=[unk_token], ) vocab.set_default_index(vocab[unk_token]) return vocab  _tsv_iterator(data_path, ngrams, column_indices): # Spacy has novel tokenizer</pre>
	<pre>tokenizer = get_tokenizer("basic_english") with io.open(data_path, encoding="utf8") as f:     reader = unicode_csv_reader(f, delimiter="\t")     for row in reader:         row_iter = [row[i] for i in column_indices]         tokens = " ".join(row_iter)         yield ngrams_iterator(tokenizer(tokens), ngrams)</pre>

**Custom Modules - Source Code** 

A number of custom code was generated to support this experiment.

datasets.py

from typing import Callable, List

 $\textbf{from} \ \text{torch.utils} \ \textbf{import} \ \text{data}$ 

import io

import torch